A new Concept for System Communication

Christian Heller <christian.heller@tu-ilmenau.de>, Torsten Kunze <info@torstenkunze.de>,
Jens Bohl <info@jens-bohl.de>, Ilka Philippow <ilka.philippow@tu-ilmenau.de>

Technical University of Ilmenau
Faculty for Computer Science and Automation
Institute for Theoretical and Technical Informatics
PF 100565, Max-Planck-Ring 14, 98693 Ilmenau, Germany
http://www.tu-ilmenau.de, fon: +49-(0)3677-69-1230, fax: +49-(0)3677-69-1220

Abstract

This paper introduces an improved architecture for system communication. The architecture is based on the Translator pattern which is derived from existing design patterns. Hierarchical abstraction and ontologies are used to combine these basic patterns and to merge their advantages into one, domain- and language-independent software framework.
This conceptual framework, called Cybernetics Oriented Programming (CYBOP), has its roots in the layered architecture pattern and is characterized by flexibility and extensibility. It helps to structure software as well as to keep it maintainable. A Component Lifecycle ensures the proper startup and shutdown of any systems built on top of CYBOP.
Great influence was exerted by the biological model of information processing in the human brain. It provided the idea of a seemless integration of communication paradigms and persistence mechanisms. Overcoming the classical scheme of thinking in terms of Domain, Frontend, Backend and Communication, this architecture treats them all similar, as passive data models which can be translated into each other – as opposed to the classical approach that unnecessarily complicates their design.
The practical proof of this combined architectural approach was accomplished within an (ongoing) effort to design and develop a module called ReForm, for the Open Source Software (OSS) project Res Medicinae. The main task for this module is to provide a user interface for printing medical forms. It was used to examine the communication between modules and to find a structure for effective implementation and easy expansion.
Keywords. CYBOP, Design Pattern, Hierarchy, Ontology, Translator, Assembler, Mapper, Communication, Backend, Persistence, Frontend, User Interface, Res Medicinae

1 Introduction

1.1 Problem Analysis

Quality of software is often defined by its maintainability, extensibility and flexibility. In the past decades, *Procedural (Structured) Programming*, then *Object Oriented Programming* (OOP) and more recently *Component Oriented Programming* (COP) provided a number of helpful paradigms to help to achieve these goals. A major improvement was the extension of data *Type* to *Class*, owning inheritable properties and methods.
But still, design problems are evident, in nearly every software product. It is the *false combination and grouping* of classes that still keeps us away from clear and effective solutions. As a system grows, the interdependencies between its single parts grow with. Why does this happen? Simply because a clear architecture is missing. Even if developers really try to follow a such – on some point in the software's life, compromises have to be made due to unforeseen requirements and dependencies:

- *Interfaces* are used to realize new properties by *multiple inheritance* (*Mix-In*)
- *Static manager objects* accessible by any other objects in a system are introduced
- New software layers are plugged in with *varying mechanisms*
- *Redundant code* needs to be written (per copy & paste), to avoid too many unwanted interdependencies

Can this be avoided at all and if, then how? The author's opinion is *yes* and the new concept introduced in this document can hopefully contribute a small part to this, and show ways out of the misery.

1.2 Architecture Context

To stepwise approach a solution, the problem needs to be placed in the right context, that is the *Physical* as well as *Logical System Architecture*.

Figure 1 shows a typical *Information Technology* (IT) environment (physical architecture). There is a central *Application Server* that is accessed by *Application Clients* or over *Web*. In addition, it can be addressed by *Local Processes* (running on the same machine) and, most importantly, *Human Users* who control the system. The application server itself can become a client by interacting with a *Database System* or other *Remote Servers*.

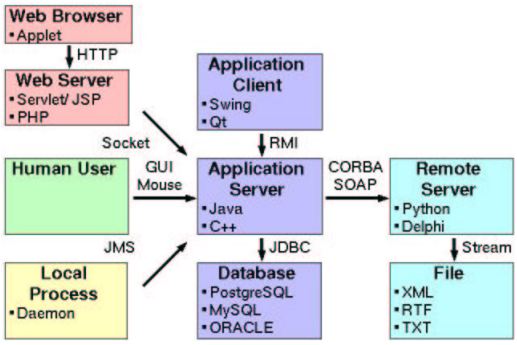

Fig. 1. Information Technology Environment

Each of these kinds of communication makes use of its own interaction mechanism. This paper will concentrate on the three (plus one) most important ones:

- Database (*Backend*)
- Remote Server
- Human User (*Frontend*)
- Business Knowledge (*Domain Model* in the Application Server itself)

A system to be capable of interacting via all of the above-mentioned kinds, needs to have implemented the corresponding mechanisms in its software (logical architecture). Figure 2 shows a possible inner software structure of a modular (layered) system implementation.

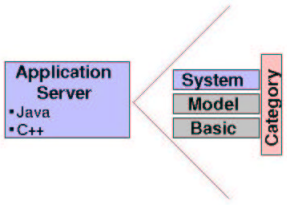

Fig. 2. Inner Software Structure

1.3 Design Pattern and Framework

One well-known way to create a layered system with clear architecture and only few interdependencies is the use of *Software Patterns* (often divided into *Design-*, *Architecture-* and other patterns). They help recognizing recurring structures for application on similar problems. Another, closely related technique are *Frameworks*. They can help prevent code duplication and development efforts. Both concepts – frameworks and patterns – depend on each other and provide higher flexibility for software components [Pre94]. This paper is not about frameworks but its solutions are extracted and used in one called *Cybernetics Oriented Programming* (CYBOP) [Pro04b]. Its main concept is based on the hierarchical structure of the universe [CH03]. This very simple idea can perfectly be mapped on software systems.

1.4 Document Structure

In the course of this paper, basic design patterns are introduced in section 2 and placed into the greater architecture context in section 4. Section 3 provides some knowledge to achieve this, that is to apply hierarchies to obtain ontologies which finally help to combine and merge the design patterns. The biological considerations of section 5 greatly contribute to the resulting architectural pattern *Translator* which gets introduced in the context of system communication, in section 6. Practical proof is given in section 7 which briefly describes the *ReForm* software module, before the final summary in section 8.

2 Basic Patterns

2.1 Data Mapper

Originally, the communication approach of CYBOP was based on the *Data Mapper* pattern (figure 3).

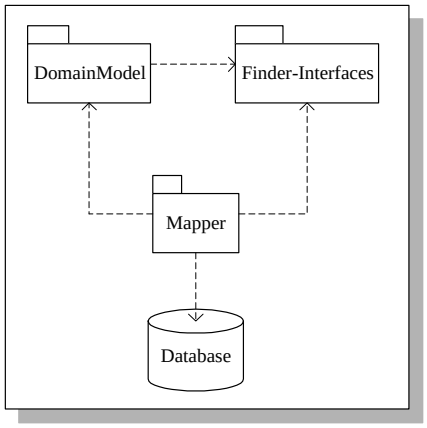

Fig. 3. Data Mapper Design Pattern [ea02]

It is part of Martin Fowler's pattern collection called *Enterprise Application Architecture* [ea02]. The most important idea of this pattern is to abolish the interdependency of domain (knowledge) model and database (persistence).
The arrows in figure 3 indicate the direction of dependency. Each domain model class knows its appropriate

persistence finder interface but does not know their implementation, i.e. how data are actually retrieved from the database. The data mapper implementation is part of the mapping package that implements all finders and maps all data of the received result set to the special attributes of the domain model objects. There is no need for the domain model to know where the database is located or how to get the data – and also not how to map the entity-relationship model data.
If all these things are done by the corresponding data mappers now, why shouldn't it be possible to get such a mapping package for persistence media of any kind, no matter which communication paradigm (File Stream, JDBC with SQL) is used? Users would have a number of persistence mechanisms to dynamically choose from; developers would not have to implement the same mechanisms again and again for each new module (application) – leading to clearer code with greatly reduced size.
This functional code separation would make it easy to develop a complicated domain model and update it later, if necessary. The data mapper package could contain special parts for local storage in a file system, in various file formats such as XML, CSV, TXT etc. (whether it makes sense or not to store domain data in a pure text file), for a number of (relational) databases (PostgreSQL, MySQL) and so on. Each of those specialised parts would know how to communicate with its appropriate persistence medium and only with it. They would all include a specialised mapper class, called *Translator* in CYBOP, which translates the data from the domain model to the model of the corresponding persistence mechanism.

2.2 Data Transfer Object

It is a well-known fact that many small requests between two processes, and even more between two hosts in a network need a lot of time. The local machine with two processes has to permanently exchange the program context and the network has a lot of transfers. For each request, there is at least a necessity of two transfers – the question of the client and the answer of the server.
Transfer-methods are often expected to deliver common data such as a Person's address, i.e. surname, first name, street, zip-code, town etc. These information is best retrieved by only one transfer-call. That way, the client has to wait only once for a server response and the server does not get too many single tasks. In the address-example, all address data would best be packaged together and sent back to the client.

And that is exactly what the Data Transfer Object pattern (figure 4) proposes a solution for: A central *Assembler* class takes all common data of the server's domain model object and assembles them together into a special object called *Data Transfer Object* (DTO) which is a flat data structure. The server will then send this DTO over network to the client. On the client's side, a similar assembler takes the DTO, finds out all received data and maps (disassembles) them to the client's domain model. In that manner, a DTO is able to drastically improve the performance in communications.
Comparing with the Data Mapper from chapter 2.1, the assembler's task of translating between data models seems quite similar, if not the same. Hence, why shouldn't it be possible for inter-system communications over network to use a *Translator* similar to the one for persistence? This translator could provide special parts for assembling different types of DTOs, independent from which communication protocol/language (Sockets, RMI, JMS, CORBA, SOAP etc.) is used.

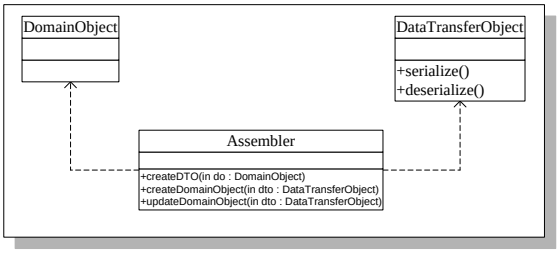

Fig. 4. Data Transfer Object Design Pattern [ea02]

2.3 Model View Controller

After having had a closer look at common design patterns for persistence and communication, this section finally considers the so called *Frontend* of an application which is mostly realized in form of a graphical user interface.
Nowadays, the well-known *Model View Controller* pattern (figure 5) is used by nearly all standard applications. Its principle is to have the *Model* holding domain data, the *View* accessing and displaying these data and the *Controller* providing the workflow of the application by handling any signals (events/ actions) appearing on the view.

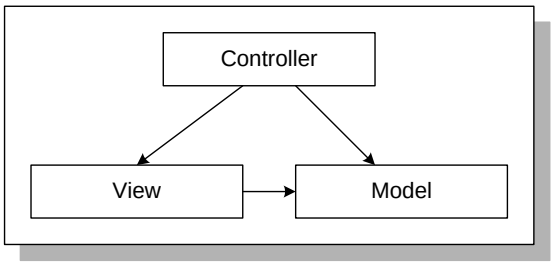

Fig. 5. Model View Controller Design Pattern

Since the view (graphical user interface) serves as means of communication between a software system (application) and its user (Human Being as system), the view is in fact just another type of communication model that should be assembled by a special translator.
Because there are many ways in which domain data can be displayed, different user interfaces can exist. Each of them has to have its very own translator item that knows how to map data both ways, from the domain model to the user interface model and vice-versa.

3 Hierarchy and Ontology

Section 2 explained three design patterns that are widely used in software architectures. It has shown similarities between them and raised the question if they could possibly be merged into just one pattern, called *Translator*, what will be described in section 4. Yet before, this section will demonstrate how the principle of *Hierarchy* may be applied to obtain an *Ontology*.

3.1 Association Elimination

An *Electronic Health Record* (EHR) will serve as example domain model whose class structure is shown in part in figure 6. It consists of numerous parts whereof at least two will be of type *Address* and *Problem*, respectively. Following the *Episode-based EHR* recommendation [HW98], *Problem* may consist of *Subjective* and *Objective*. All these associations between classes are needed to navigate through the domain model.

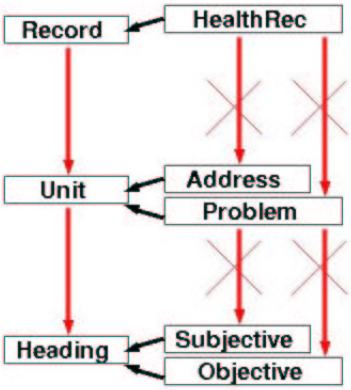

Fig. 6. Parent- eliminate Child Associations

A frequent design decision in object oriented programming is to sum up common properties of sub classes by introducing a common super class. It is not only properties, but also the *Granularity* of objects that can lead to the creation of a super class. The OpenEHR project [Pro04a] suggests to let the above-mentioned classes inherit from the more coarse-grained super classes *Record*, *Unit*, *Heading* and others.
Whichever reason – once the super classes are there, they should be associated similarly to their sub classes, that is in the same direction, using unidirectional dependencies. Afterwards, all associations between sub classes become superfluous as every sub class can reach its sibling across their parent classes' association (figure 6).
Here a short Java code example for how the *HealthRecord* may retrieve a reference to *Address*:

```
Address a = (Address) get("address");
```

HealthRecord inherits the *get* method from its super class *Record*. *Record* holds many instances of type *Unit* and differing sub types. The *get* method delivers back an object that still needs to be down-casted to the expected sub type *Address*.
The definition of classes, their dependencies (defined by associations) and granularities (defined by inheritance) in a software system results in several layers of classes of common granularity, as shown in figure 7. These layers are often called *Ontological Level* as they form an *Ontology* (see section 3.2).

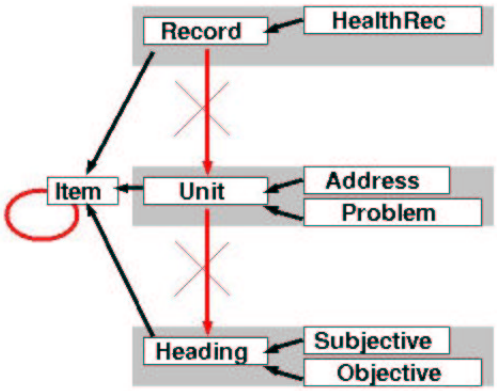

Fig. 7. Ontological Levels and Item Container

Continuing the structure process of introducing more and more common, coarse-grained or fine-grained super classes, the development culminates in one top-most super class of all other classes in the system, which this paper calls *Item*. It is as general as the *java.lang.Object* class for the Java class library, only that it additionally represents a container that can store objects of any type, as explained in [CH03]. In other words, *Item* provides the meta functionality of a container behaviour to *all* other classes.

3.2 Ontology

Manifold definitions of the word *Ontology* exist. They come from philosophy, metaphysics, information technology and others – too many to list here. This document uses its own, adapted definition and considers an ontology to be *a strict hierarchy of abstract items, organized in levels of growing granularity, that are solely unidirectionally related*. It such represents a systematic description of complex domain contexts.

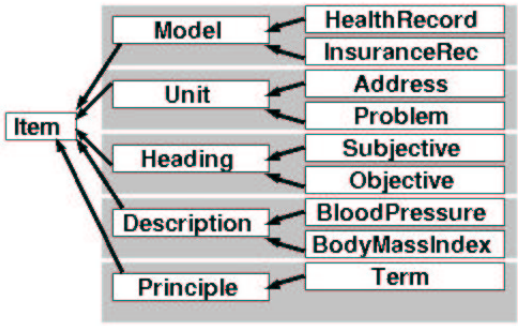

Fig. 8. Electronic Health Record Ontology

Figure 8 shows one possible ontology of an electronic health record, as described in the previous section.

4 Logical Architecture

This section will sort the design patterns of section 2 into the layered architecture of a standard application. Afterwards, the hierarchical principles of section 3 are applied to simplify and merge the design patterns which will lead to an ontology.

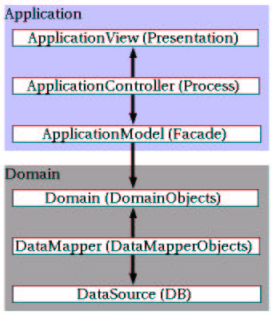

Fig. 9. Layered Architecture

A state-of-the-art software system consists of a layered architecture similar to the one shown in figure 9. The startable *Controller* process creates the whole application tree, to which belong the *View* (as user interface), the *Model* (providing data to the view and as facade to remote servers) and the *Domain* with its database *Mapper* layer.
It is not difficult to figure out where the basic patterns of section 2 fit in here (figure 10): The *Model View Controller* pattern determines the classes to interact with a human user via the *View* (sometimes called *Presentation Layer*); the *Data Mapper* pattern provides necessary classes and an *Entity Relationship Model* (ERM) to connect to a persistence medium such as a database; the *Data Transfer Object* (DTO) pattern, finally, serves as means of communication with remote servers.

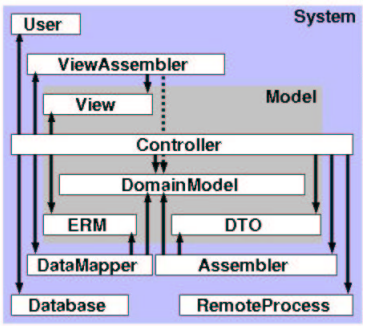

Fig. 10. Layered Architecture with Basic Patterns

For all three kinds of communication, there is a:

- System (HumanUser, DataBase, RemoteServer)
- Model (View, ERM, DTO)
- Translator (ViewAssembler, Mapper, DTOAssembler)

Realizing this, it is easy to create ontological layers by adding one common parent class for systems, models and translators each, which leads to a much clearer architecture (figure 11). The common properties of all sub classes are merged into their corresponding super class.

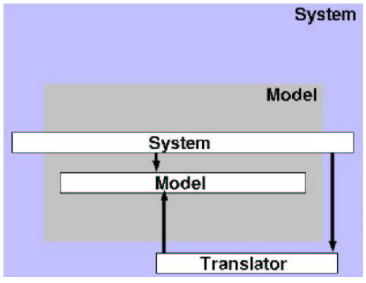

Fig. 11. Layered Architecture with merged Patterns

5 Biological Reflections

The previous sections have shown how existing patterns for communication can be merged into one common system architecture. All of these design patterns suggest their very own communication paradigm which cannot be used anymore in the new, merged *Translator* architecture. Therefore, a new way for system interaction needs to be found.

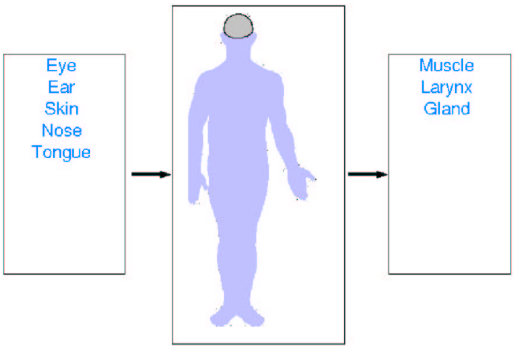

Fig. 12. Human Being as System of Models (Brain)

Following the CYBOP approach, nature – in our case the Human body – will be considered next. Humans have organs responsible for information input and output (figure 12). In between input and output, the information is processed by the brain that contains a specific abstract model of the surrounding real world. The human brain consists of several regions, each being responsible for a special task, such as the optical region for seeing or the cerebral cortex for actual information processing which possibly leads to awareness.

The following example demonstrates a typical information (signal) processing procedure (technical names were used instead of biological ones in figure 13; the terms *Mapper* and *Assembler* are converted and merged into the term *Translator*):
One human *System* wants to send another human *System* a message. It decides for an acoustical *Signal*, formulates a sentence and talks to the other human *System* (*handle* method). The other human receives the *Signal* across its ear organ (*Keyboard*, *Mouse*, *Network*). The *Signal* is then forwarded to the receiver's brain (*Controller*) where a special *Region* responsible for acoustics (*Translator*) translates (*decode* method) the data (*DataTransferModel*) contained in the *Signal* and sorts them into the human's abstract model of the surrounding real world (*DomainModel* or *KnowledgeModel*, respectively). Processing of the signal happens in the cerebral cortex of the brain (*Processor*). If the addressed listener wants to send an answer *Signal*, it may do so by triggering a muscle reaction. For this to happen, the motoric brain region (*Translator*) needs to

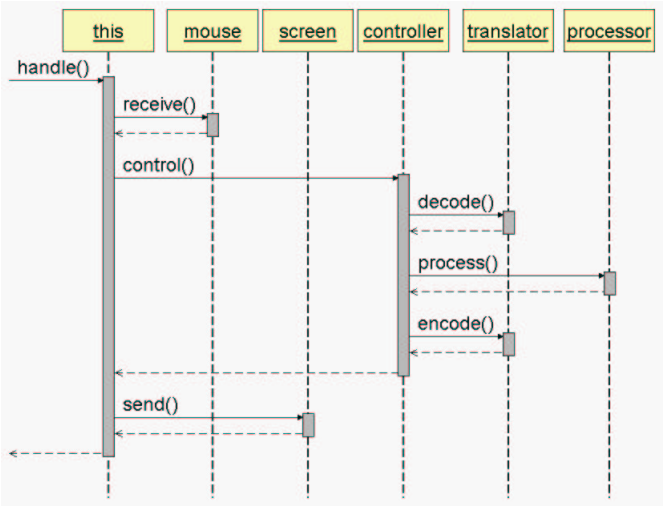

Fig. 13. Signal Processing as UML Sequence Diagram

translate (*encode* method) abstract model data (*DomainModel*) into a special communication model (*UserInterfaceModel*) for the answer signal. Finally, the answer signal will be sent as muscle action (data display on *Screen*).

6 System Models

So far, the paper has elaborated on the statics (section 4) as well as the dynamic side (section 5) of the proposed *Translator* pattern. This section will finally show the overall results in a number of architecture diagrams.

6.1 Translator Pattern

As could be seen in section 5, there is always a *Translator* that is able to map domain model data to communication model data (*encode* method) and back (*decode* method). Depending on which communication medium is used, different translators need to be applied (figure 14).

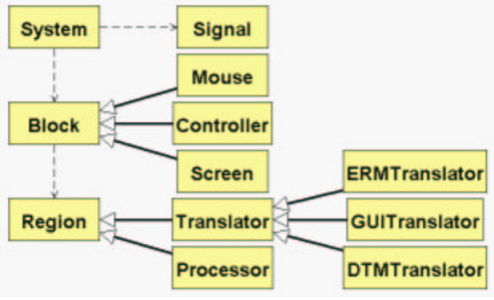

Fig. 14. Translator Classes in a UML Class Diagram

Every system has exactly one domain model but communication models of arbitrary type can be added anytime (figure 15). Every translator knows only how to translate between the domain model and a special communication model. Direct translation between communication models is forbidden as it would break the flexibility of the whole framework. In other words, translations always have to be done *via* the domain model.

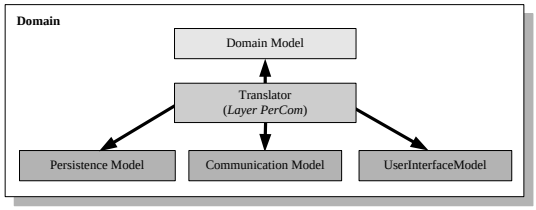

Fig. 15. Translator accessing various Models

All typecasted programming languages already contain an *Ontology*! These primitives represent the lowest layer in an ontology or in other words, the last level of abstraction in software. That is also where *Terminologies* (that are mostly mentioned in conjunction with ontologies) come in. Basically, these are sorted collections of terms (strings) but not further elaborated here.

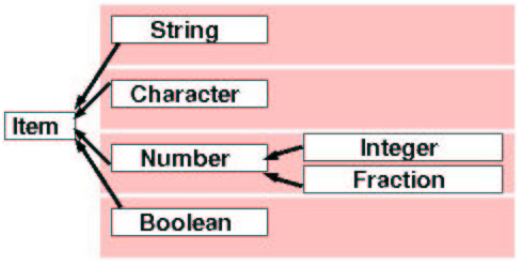

Fig. 17. Basic (Language) Ontology

6.2 Ontology Framework

When placing the translators of figure 14 into the greater system architecture context, a *System Ontology* as shown in figure 16 may be retrieved. It contains the new *Translator* as sub class of *Region*, input/ output devices as sub class of *Block*, *Module* (*Application*) and *User* as sub class of *System* and further parts which are not the topic of this paper. For the ease of understanding, the biological counterparts have been added on the right side of the figure. Specialized translators may be derived as sub class of the one shown in figure 16.

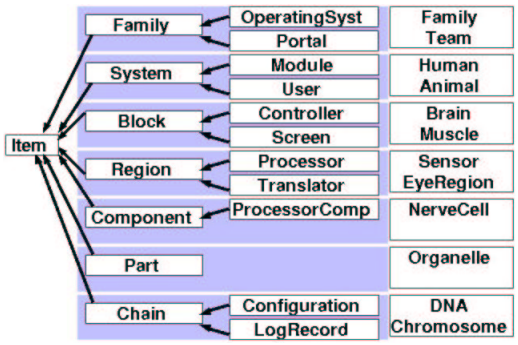

Fig. 16. System Ontology

To complete the list of important ontology models, figure 17 gives an overview of language-integrated types (commonly called *Primitives*). And as a matter of fact:

Putting the three ontologies *Basic*, *Model* and *System* that were introduced in this paper together, results in the CYBOP architecture of figure 18. All ontologies base on the *Language Ontology*. A system built after the *System Ontology* model (in this paper the example of an *Electronic Health Record* application) may access one ore more *Model Ontologies* (in the example the health record domain model). All dependencies are unidirectional.

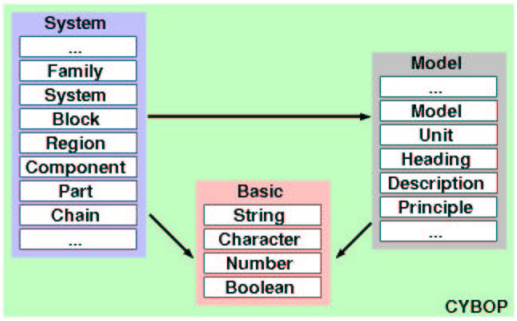

Fig. 18. CYBOP Ontology Framework

6.3 Consistency

The described models are highly flexible and extensible and absolutely transparent to the user (developer). S/he will not know whether the current communication is with the local file system, a database or a remote process on another machine.

However, this transparency causes a number of problems. Surely, the most common question is how to ensure consistency, security and minimum redundance? The following two paragraphs give an answer to the first part of this question – consistency and uniqueness of data sets. Maximizing security and minimizing redundancy have to be analysed in future works.

Object ID (OID) Most database systems provide an own algorithm to generate primary keys for the tables. But the applications that use our communication architecture shall also be able to work if a database server is not reachable, e.g. due to a network failure. Thats why the keys are generated locally, by each application. Based on the assumption that every host in a network has a network card, it thereby has a unique internet address. This number is concatinated with an exact time stamp (nanoseconds). That is why the OID is unique in the global network and unique in time.
The proposed approach uses the OID as file name for local storage and the same OID as primary key in the main table of the database. Therewith, both models can be mapped to each other. Of course, it is necessary to avoid overwriting of new data in the database. If, for example, a network connection is cut and a little later, one wants to get data from the local files and write them up in the restored central database, it has to be made sure that nobody else has modified the data during the offline-time. That is why there is another technique to ensure this – the time stamp.

Time Stamp Most database developers will know this technique. Each table has a separate column for storing the time at which the data were written into this table. If someone requests information from the database, the time stamp is delivered as well. After modifying the data, they have to be written back into the database. At this time, both timestamps (the one in the database table and the one delivered before) are compared. If there is a difference, the data were modified by another user. Then, one has to care about the update without overwriting the new data in the table.

7 Physical Architecture

This section wants to give practical proof of the theoretical models described before. It first introduces the project *Res Medicinae* in whose frame the software was written. Afterwards, two solutions of a physical architecture, *Two-Tier* and *Three-Tier* are explained.

7.1 Res Medicinae

The practical background for the application of CYBOP is *Res Medicinae* [pro04c]. A modern clinical information system is the aim of all efforts in this project. In the future, it shall serve medical documentation, laboratory data, billing etc.
Res Medicinae is separated into single modules solving different tasks. One module in which the CYBOP communication concepts were applied is *ReForm* (figure 19). It offers a medical form that can be filled in and printed out. Since one of the main reasons to implement this module was the testing and proof of the new persistence and communication concepts, it includes a dialog for choosing the communication protocol or persistence mechanism, respectively. This is the only remaining part where users have to care about the underlaying techniques. They also have to decide whether to use the local file system via *Extensible Markup Language* (XML) format or to store the data in a central database. In the future, an XML file format may as well be used for remote communication, e.g. via *Simple Object Access Protocol* (SOAP).
Because of the component-based design of *Res Medicinae*, it is possible to start more than one instance of *ReForm* at the same time. In this way, the data exchange between modules can be tested. A module *X* looks for another registered module *Y* at the naming service of *Remote Method Invocation* (RMI), *Common Object Request Broker Architecture* (CORBA) or some other. *X* gets the address of the remote service *Y* (depending on the communication mechanism). The stub and skeleton of *X* and *Y* marshal, send and unmarshal the data for further working.

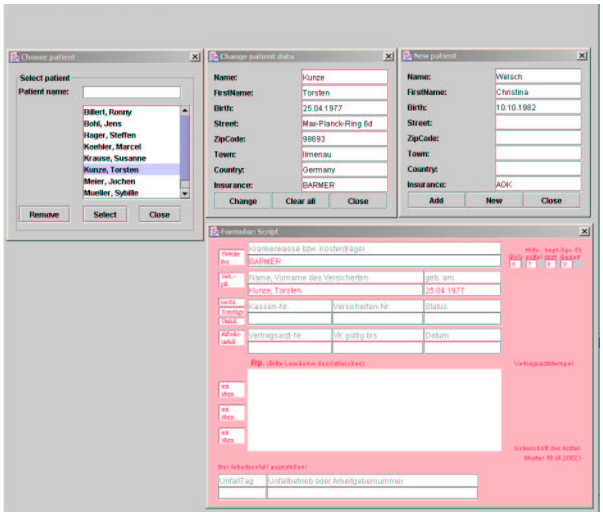

Fig. 19. ReForm Module

7.2 Two Tier Architecture

The proposed CYBOP communication architecture is currently implemented in form of a *Two-Tier* architecture (figure 20).

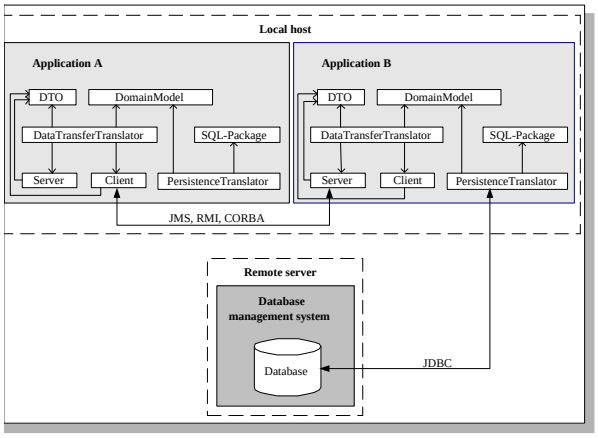

Fig. 20. Two Tier Architecture

It shows how two autarchic components (Application A and B) intercommunicate and save their domain data in different ways. Each component fulfills a special task and works as a client as well as a server. One can recognize the two patterns *DataMapper* and *DTO*.
If a client requests some data from another component, the central object *DataTransferTranslator* collects all needed information from the *DomainModel* and encodes (packs) them into one *DataTransferModel*. Now, the *Server* object can send this *DTO* back to the requesting client component. On the other side of the wire, the *Client* object receives the *DTO*, a *DataTransferTranslator* decodes (unpacks) the data and writes them into the *DomainModel*.
In this example, the two components are located on the same host. It is also possible to distribute them. Therefore, each component is also able to communicate with other components that are situated somewhere in the network. The arrows in applications indicate the dependencies between the single architectural elements, whereas the outside arrows show the communication between components and database server.
All data storing operations are hidden in a special *PersistenceTranslator* like the one shown in figure 20, on the example of a database. The SQL statements were placed in a separate package. If there is the need for getting information from a database, the translator uses the statements of the *SQL Package* and maps data of the result set to the *DomainModel*.

7.3 Three Tier Architecture

To provide a more comfortable structure than the typical *Two-Tier* architecture as shown in section 7.2, there is the necessity of a *Three-* or *Multi-Tier* Architecture. If, for example, the location of the database server was changed then, in a *Two-Tier* architecture, all clients would have to be updated. Figure 21 makes a proposition to solve this problem.

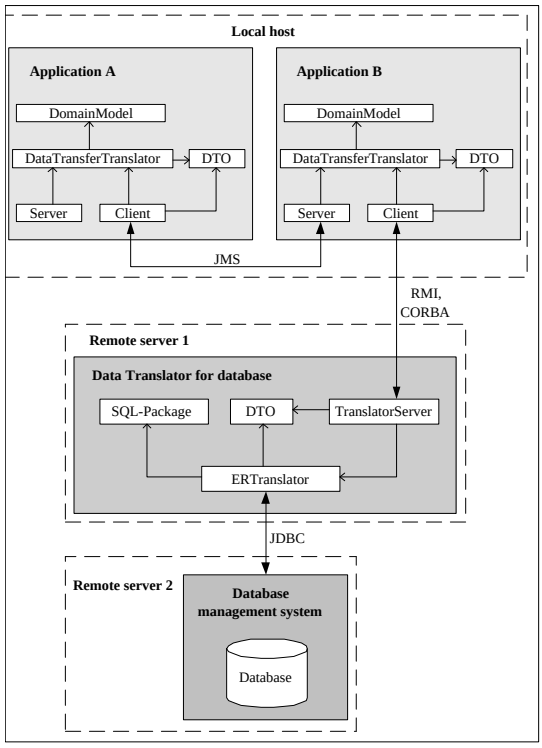

Fig. 21. Three Tier Architecture

8 Summary

Major research objectives are to find concepts and principles to increase the reusability of software, their architectures as well as the resulting code. The aim of this work was to find an architecture that simplifies and unifies the implementation of any kind of communication mechanism.
Persistence, remote communication and user interface mechanisms have common properties. Classical system architectures treat them as *Backend*, *Data Transfer* and *Frontend* and use different methods and design patterns (*DataMapper*, *DataTransferObject*, *ModelViewController*) to implement them.
This paper proposed to sum up their common properties

and behaviour and to merge them into just one communication pattern called *Translator*, thereby avoiding redundant parts. The new pattern required a new communication paradigm and this paper described one that follows the information processing procedure of the human brain. Finally, the pattern was integrated into a greater system context using ontologies. The proposed ontology framework consists of a *Basic*, a *Model* and a *System* Ontology and seems to be a good solution for the implementation of highly flexible, easily extensible and maintainable source code. The interdependency of domain data, persistence layer, communication layer and user interface is abolished.

The time needed to create such an architecture (like in form of the CYBOP framework) is clearly more than for the classical way. But once the architecture is there – it can save a tremendous amount of time when deriving modules being capable of communicating across various mechanisms at once. Due to its flexibility and low dependencies, it also ensures that extensions (e.g. new communication mechanisms) and modifications can be done anytime later without destroying already existing solutions.

9 Acknowledgements

Our special thanks go to all Enthusiasts of the Open Source Community who have provided us with a great amount of knowledge through a comprising code base to build on. We'd also like to acknowledge the contributors of *Res Medicinae*, especially all medical doctors who supported us with their analysis work [KH04] and specialised knowledge in our project mailing lists.

References

[CH03] CHRISTIAN HELLER, JENS BOHL, TORSTEN KUNZE ET AL.: *Flexible Software Architectures for Presentation Layers demonstrated on Medical Documentation with Episodes and Inclusion of Topological Report*. Journal of Free and Open Source Medical Computing (JOSMC), June 2003. http://www.josmc.net.

[ea02] AL., MARTIN FOWLER ET: *Patterns of Enterprise Application Architecture (Information Systems Architecture)*. Addison-Wesley, Boston, Muenchen, 2001-2002. http://www.aw.com.

[HW98] HENK WESTERHOF, DUTCH COLLEGE OF GENERAL PRACTITIONERS, UTRECHT: *Episodes of Care in the New Dutch GP Systems*. Primary Health Care Specialist Group Annual Conference Proceedings, Cambridge, September 1998. http://phcsg.ncl.ac.uk/conferences/cambridge1998/westerhof.htm.

[KH04] KARSTEN HILBERT, CHRISTIAN HELLER, ROLAND COLBERG ET AL.: *Analysedokument zur Erstellung eines Informationssystems fuer den Einsatz in der Medizin*. The Res Medicinae Free Software Project, 2001-2004. http://www.resmedicinae.org/model/analysis.

[Pre94] PREE, W.: *Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design*. *Proceedings of ECOOP '94*, 150–162, 1994.

[Pro04a] PROJECT, OPENEHR: *Open Electronic Health Record (OpenEHR), formerly Good Electronic/ European Health Record (GEHR)*, 2000-2004. http://www.openehr.org.

[Pro04b] PROJECT, THE CYBOP: *Cybernetics Oriented Programming (CYBOP)*, 2002-2004. http://www.cybop.net.

[pro04c] PROJECT, THE RES MEDICINAE: *Res Medicinae – Medical Information System*, 1999-2004. http://www.resmedicinae.org.